Energy-Efficient Pipelined Bloom Filters for Network Intrusion Detection

Ilhan Kaya and Taskin Kocak
Department of Electrical and Computer Engineering
University of Central Florida
Orlando, FL 32816
e-mail: {ikaya, tkocak}@cs.ucf.edu

***Abstract*— Software-based detection techniques are commonly used to identify the predefined signatures in network streams. However, the software-based techniques can not keep up with the speeds that network bandwidth increases. Hence, hardware-based systems have started to emerge. Bloom filters are frequently used to identify malicious content like viruses in high speed networks. However, architectures proposed to implement Bloom filters are not power efficient. We propose a new Bloom filter architecture that exploits the well-known pipelining technique. Through extensive power analysis we show that pipelining can reduce the power consumption of Bloom filters up to 90 %, which leads to the energy-efficient implementation of network intrusion detection systems.**

I. Introduction

Nowadays, there are plenty of software programs installed to keep computer systems clean. Many of them are used in implementation of network intrusion detection systems(*NIDS*). These programs have to identify the malicious content such as internet worms and viruses in network packets. Applications providing intrusion detection, virus prevention, and content filtering have not kept pace with the increase in network speeds since they lack hardware functionality supporting them. There is a need to scan entire network packets bit by bit to determine predefined signatures for viruses and worms.

Before the packets are processed by the upper OSI layers, malicious packets has to be dropped by a device. Bloom filters [2] are used in such devices to match strings, like snort rules [11]. Bloom filters have been used for many network applications like web cache sharing [6], resource routing [5], string matching [1] [7]. A hardware system, consisting of Bloom filters to detect malignant content, is described in [7]. A detailed survey of Bloom filters for networking applications can be found in [3].

Although Bloom filters have found wide spread usage in networking applications, they are not conservative in terms of power. In order to decrease power consumption of Bloom filters, we employ pipelining technique in the architecture of Bloom filters. The new type of Bloom filters are called as *pipelined Bloom filters*. In this paper, we propose a hardware system consisting of pipelined Bloom filters as an energy-efficient network intrusion detection system. Furthermore, after mathematical analysis, intrusion detection system is shown to be much power efficient than the one consists of regular Bloom filters so far.

II. Pipelined Bloom Filters

In this section, first we introduce Bloom filters, and then we go on to explain pipelined Bloom filters architecture. Power analysis will follow in a comparative manner.

A. Bloom Filters

A Bloom filter is a data structure that stores a given set of signatures. In addition, it provides efficient membership queries on the signature set. Two operations on the Bloom filters are defined as *programming* and *querying*.

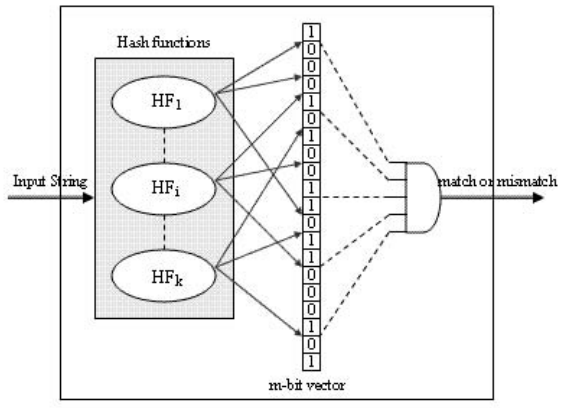

Fig. 1. A Bloom Filter with k hash functions

A typical Bloom filter is illustrated in Fig. 1. First, a Bloom filter computes *k* many hash functions on each member of the signature set. Consequently, it sets the bits on the *m-bit* long lookup vector at the indices pointed by hash values. As for the querying, the Bloom filter computes the same hash functions used in the programming for each input. Then it looks up the bit values located on the offsets (computed hash values) on the bit vector, and, if it finds any bit unset at those addresses, it declares the input string to be a nonmember of the signature set, which is called a *mismatch*. Otherwise, if it finds all the bits are set, it concludes that input string may be a member of the signature set with a certain probability (*false positive probability*). This is called a *match*.

In [7], the false positive probability f is calculated by,

$$f = \left(1 - e^{\frac{-nk}{m}}\right)^k \quad (1)$$

1-4244-0355-3/06/$20.00 (c) 2006 IEEE

where *n* is the number of signatures programmed into the bloom filter, *m* is the length of the lookup vector, and *k* is the number of hash functions used to implement the Bloom filter. In addition, the authors find an optimum value of the number of hash functions, *k =35*, for an average number of bits allocated to per signature $\frac{m}{n} = 50$. Furthermore, they show that the false positive probability is on the orders of 10^{-11}.

A single Bloom filter uses *k* many hash functions in order to make a decision on the input. Hence the power consumption of a Bloom filter shown in Fig. 1 is a summation of the power consumptions of each of the hash functions, P_{H_i}, with the lookup operation, P_L, followed, plus an AND operation.

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_i} + P_L) + P_{AND} \quad (2)$$

Power consumption of an AND gate is ignored hereafter, since it is minimal compared to the power used by the hash functions. The power consumption equation for a single Bloom filter simply becomes the total power used up by the hash functions and the power consumed by querying the m-bit vector for each hash value.

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_i} + P_L) \quad (3)$$

Hash functions used in the Bloom filters are generally of type *universal hash functions* [4]. The performance of universal hash functions are explored by Ramakrishna et al [9]. The power consumptions of different hash function implementations in hardware have been measured by Yuksel [12]. Yuksel makes use of the multi-hashing [10] technique. Multi-hashing divides the input into smaller pieces. Different universal hash functions from the same family applied to these input pieces. Eventually results are concatenated to form the desired hash value without increasing the hash collision probability.

Yuksel has come up with different hash function implementations over word sizes of 64 bits, 32 bits, 16 bits. Amongst the different hash functions analyzed in [12], the most power efficient hash function for a fixed frequency is *Weighted NH Polynomial with Reduction, WH*. This hash function is a derivative of the *Universal Hash functions*. Hence we use 16-bit, 32-bit, 64-bit WH family of hash functions in the Bloom filters whenever hash functions are able to process the input fed to the system. A WH family of hash functions can process input as twice as longer than the word size used in implementation, for instance a 16-bit WH can process up to 32-bit strings including 32-bit input.

In order to compare the power consumption of regular Bloom filters to that of a pipelined Bloom filter, we use 16-bit implementation of hash functions. All of the *k* many hash functions are of type 16-bit WH hash functions, so Equ. 3 becomes

$$P_{BF_{regular}} = \sum_{i=1}^{k} (P_{H_{i(WH_{16})}} + P_L) = k.(P_{WH_{16}} + P_L) \quad (4)$$

B. Pipelined Bloom Filters

Basically, a pipelined Bloom filter, as shown in Fig. 2, consists of two groups of hash functions. The first stage always computes the hash values. By contrast, the second stage of hash functions only compute the hash values if in the first stage there is a match between the input and the signature sought.

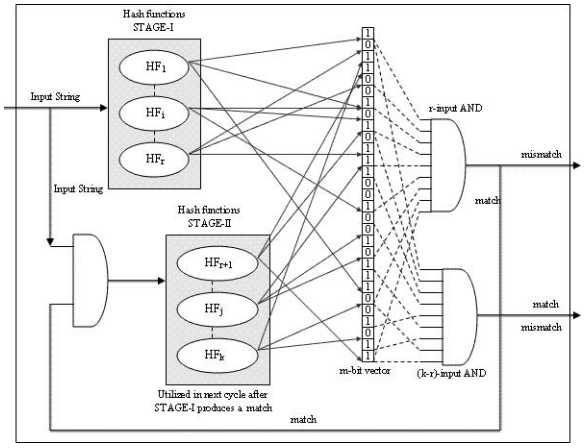

Fig. 2. A 2-Stage pipelined Bloom filter

The pipelined Bloom filters will have the same number of hash functions as the regular Bloom filters. A pipelined Bloom filter exploits the virus free nature of the network traffic at most of the time. At worst, it will operate like a regular Bloom filter, which uses all of the hash functions before making a decision on the type of the input. However, most of the time the first group of hash functions will result in a mismatch. The advantage of using a pipelined Bloom filter is if the first stage catches a mismatch, there is no need to use the second stage in order to decide whether the input string is a member of the signature set.

By following a similar analysis of [8], we begin to derive the total power consumption of the pipelined Bloom filter. We assume that the hash functions used in each Bloom filter is perfectly random. Overall, we have *k*-many random universal hash functions in a pipelined Bloom filter. In the first stage, *r*-many of them are utilized. The number of signatures sought in a Bloom filter is given as *n* as before.

Let us first derive the probability of match in the first stage. The probability that a bit is still unset after all the signatures are programmed into the pipelined Bloom filter by using *k*-many independent hash functions is α.

$$\alpha = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{\frac{-kn}{m}} (for\, large\, m) \quad (5)$$

where $\frac{1}{m}$ represents any one of the *m* bits set by a single hash function operating on a single signature. Then $\left(1 - \frac{1}{m}\right)$ is the probability that the bit is unset after a single hash value computation with a single signature. For it to remain unset, it should not be set by any of the *k*-many hash functions each

operating on all of the *n*-many signatures in the signature set. Consequently, the probability that any one of the bits is set is

$$(1-\alpha) \approx 1 - e^{\frac{-kn}{m}} \tag{6}$$

In order for the first stage to produce a match, the bits indexed by all *r* of the independent random hash functions should be set. So the match probability of the first stage is, represented as *p*,

$$p = \prod_{i=1}^{r} (1-\alpha) = (1-\alpha)^r \approx (1 - e^{\frac{-kn}{m}})^r \tag{7}$$

With a probability of *(1-p)* the first stage of the hash functions in the pipelined Bloom filter will produce a mismatch. Otherwise, the first stage produces a match, then the second stage is used to compare the input with the signature sought. Therefore the power consumption of a pipelined Bloom filter is given by

$$\begin{aligned} P_{BFpipeline} &= P_{1st-stage} + P\{match\} \times P_{2nd-stage} \\ P_{BFpipeline} &= \sum_{i=1}^{r} (P_{H_i} + P_L) + p \times \sum_{j=r+1}^{k} (P_{H_j} + P_L) \\ &\quad + P_{AND} \end{aligned} \tag{8}$$

By omitting P_{AND} and selecting $P_{H_{i,j}}$ are of type 16-bit WH, and substituting Equ. 7 into Equ. 8 power consumption of a pipelined Bloom filter becomes

$$\begin{aligned} P_{BF_{pipeline}} &= \sum_{i=1}^{r} (P_{H_{i(WH_{16})}} + P_L) \\ &\quad + (1 - e^{\frac{-kn}{m}})^r \times \sum_{j=r+1}^{k} (P_{H_{j(WH_{16})}} + P_L) \\ &= r.(P_{WH_{16}} + P_L) + \\ &\quad (1 - e^{\frac{-kn}{m}})^r (k-r)(P_{WH_{16}} + P_L) \end{aligned} \tag{9}$$

The power saving ratio, *PSR*, in a single Bloom filter by deploying pipelining technique can be calculated as

$$PSR = \frac{(P_{regular} - P_{pipelined})}{\mathrm{P}_{regular}} \tag{10}$$

By substituting Equ. 4 and Equ. 9 into Equ. 10, the average power saving ratio, *PSR*, is given by

$$PSR = \frac{\left(k \times A - \left[r + (1 - e^{\frac{-kn}{m}})^r \times (k-r)\right] \times A\right)}{k \times A} \tag{11}$$

where $A = (P_{WH_{16}} + P_L)$. After simplifying *A*s, *PSR*, is found out to be

$$PSR = \frac{k - r + (r-k)\,(1 - e^{\frac{-kn}{m}})^r}{k}$$

For different values of the number of bits allocated to per signature, $\frac{m}{n}$, power savings over the number of hash functions utilized in the first stage are illustrated in Fig. 3.

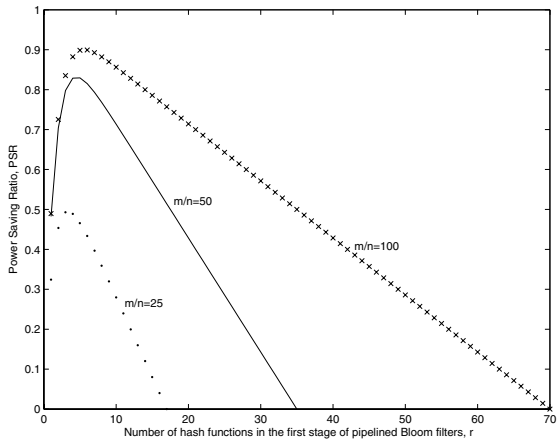

Fig. 3. Power saving ratio in pipelined Bloom filters w.r.t. number of hash functions utilized in the first stage

As it is shown in Fig. 3, the number of bits per signature, $\frac{m}{n}$, increases, the amount of power conserved in the system increases. In other words, the power saving ratio becomes larger as $\frac{m}{n}$ increases. This is because, the lesser are the number of hash functions deployed in first stage compared to the overall system, the more the power is saved. Another fact illustrated in Fig. 3 is that as the number of hash functions utilized in the first stage increases, the power saving ratio, *PSR*, first increases to an optimum *PSR* value, thereafter it drops gradually. The increase in the power saving ratio to an threshold value stems from the fact that increasing the number of hash functions in the first stage increases the probability of mismatch, thus second stage is not utilized to decide on an input. After this optimum is exceeded, *PSR* decreases steadily. If we increase the number of hash functions used in the first stage to such a degree that all hash functions in the system deployed in the first stage, there remain no power gain at all (i.e., the system behaves just like a regular Bloom filter.)

III. Power Consumption Analysis of an Intrusion Detection System based on Pipelined Bloom Filters

In this section, we analyze the power consumption of a Bloom filter based NIDS given in [7]. The system block diagram is illustrated in Fig. 4. The system consists of parallel Bloom filters programmed to detect a group of predefined signatures. Each Bloom filter operates on a different length of input. The length of the input scanned by a Bloom filter is constricted by the length of the signature sought. All Bloom filters monitor the network traffic which enters into system with a rate of one byte per cycle. System verifies whether there is a malicious content hidden in the packet payload. Power analysis of the system follows in two sections.

A. System relying on regular Bloom Filters

In this case, the system illustrated in Fig. 4 consists of regular bloom filters. Each Bloom filter operates on different length of inputs. Total power consumption of the system is

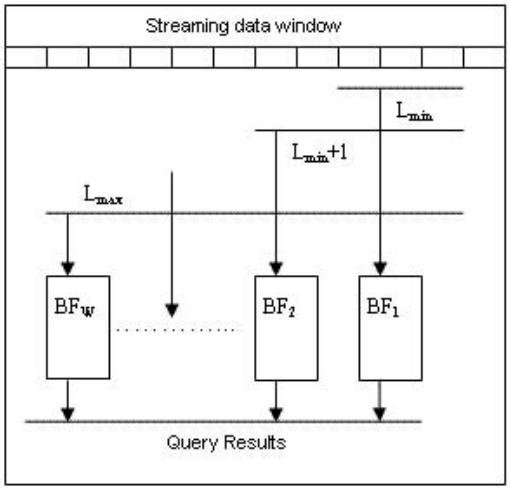

Fig. 4. System view of a Bloom filter engine [7]

the summation of the power used by individual Bloom filters. There are $(L_{max} - L_{min})$ Bloom filters in the system. L_{max} is the maximum length of input, and L_{min} is the minimum length of input processed by a Bloom filter. Hence the total power consumption is given by

$$P_{TOTAL} = \sum_{i=L_{\min}}^{L_{\max}} P_{BF_i} \tag{14}$$

where P_{BF_i} is the power dissipated through the i^{th} Bloom filter.

Let's develop the Equ. 14 as follows

$$P_{TOTAL} = P_{BF_{l_{\min}}} + P_{BF_{(l_{\min}+1)}} + P_{BF_{(l_{\min}+2)}} + ... + P_{BF_{(l\max)}} \tag{15}$$

where $P_{BF_{l_{\min}}}$ stands for the total power dissipated through the Bloom filter processing $L_{min} - long$ input. Each Bloom filter in the system takes different length inputs as pointed out before. By substituting Equ. 3 into the Equ. 14,

$$P_{TOTAL} = \sum_{i=1}^{k} P_{H_i}^{l_{\min}} + \sum_{i=1}^{k} P_{H_i}^{(l_{\min}+1)} + ... \sum_{i=1}^{k} P_{H_i}^{l_{\max}} \tag{16}$$

$P_{H_i}^{l_{\min}}$ is the power dissipation of the i^{th} hash function located inside the Bloom filter processing $l_{min} - long$ input.

We consider a system with $l_{min} = 3$ bytes and $l_{max} = 32$ bytes. We assume that lookup power is ignorable in the following analysis. We begin the analysis with calculating the power consumption of a 3-byte input processing Bloom filter.

3-byte input length is smaller than 32-bit, which can be processed by a WH_{16} type of hash function. We can divide the input into two halves. 8 bits are zero padded to the second half, which does not change the hash value computed, since bitwise AND operation makes contribution of these bits 0.

$$P_{BF_{3-byte}} = \sum_{i=1}^{k} P_{H_i(WH_{16})} = k.P_{WH_{16}} \tag{17}$$

Similarly, 4-byte processing Bloom filter uses WH-16 type of hash functions without the input padding.

$$P_{BF_{4-byte}} = \sum_{i=1}^{k} P_{H_i(WH_{16})} = k.P_{WH_{16}} \tag{18}$$

5-byte are larger than 32 bits, and can't be processed by a WH_{16}. This type of input requires WH_{32}, which can process up to 8 byte. The remaining 24 bits are zero padded to the input. Hence,

$$P_{BF_{5-byte}} = \sum_{i=1}^{k} P_{H_i(WH_{32})} = k.P_{WH_{32}} \tag{19}$$

A WH_{32} is used in the construction of 6-byte, 7-byte and 8-byte long input processing Bloom filters. Power consumption of Bloom filters corresponding to these length input is equal to the 5-byte input processing Bloom filter.

$$\begin{aligned} P_{BF_{6-byte}} &= \sum_{i=1}^{k} P_{H_i(WH_{32})} \\ &= k.P_{WH_{32}} = P_{BF_{7-byte}} = P_{BF_{8-byte}} \end{aligned} \tag{20}$$

As for the 9-byte input processing Bloom filter, a WH_{32} can't be enough without multi-hashing. Hence, WH_{64} is needed with zero padded input.

$$P_{BF_{9-byte}} = \sum_{i=1}^{k} P_{H_i(WH_{64})} = k.P_{WH_{64}} \tag{21}$$

Since a WH with a word size of 64 bit can process up to 128 bits long input, all the Bloom filters processing input in the range of 9 byte to 16 byte, including the 16 byte, can be implemented with a WH_{64}. Their power consumptions are equal.

$$\begin{aligned} P_{BF_{10-byte}} &= \sum_{i=1}^{k} P_{H_i(WH_{64})} = k.P_{WH_{64}} \\ P_{BF_{9-byte}} &= P_{BF_{10-byte}} = ... = P_{BF_{16-byte}} \end{aligned} \tag{22}$$

After 128 bit long input we have to use the multi-hashing method, and divide the input into 32 bit long and 128 bit long input, and make zero padding if necessary until input length exceeds 160. By this way, it is possible that one WH_{64} and one WH_{16} will be enough to calculate hash function over 17 to 20 byte long inputs. So the power consumption of the Bloom filters operating on 17-byte to 20 byte long input is

$$\begin{aligned} P_{BF_{17-byte}} &= \sum_{i=1}^{k} (P_{H_i(WH_{64})} + P_{H_i(WH_{16})}) \\ &= k.P_{WH_{64}} + k.P_{WH_{16}} P_{BF_{17-byte}} \\ &= P_{BF_{18-byte}} = ... = P_{BF_{20-byte}} \end{aligned} \tag{23}$$

In a similar manner, again by making use of multi-hashing and zero padding whenever necessary, hash functions inside in Bloom filters operating on range of 21 to 24 byte input, make use of one WH_{64} and one WH_{32}. All the Bloom filters calculating hash values over the inputs from 21 bytes long

till 24 bytes, consumes the same power as 21-byte long input processing Bloom filter, which is shown below.

$$\begin{aligned} P_{BF_{21-byte}} &= \sum_{i=1}^{k} (P_{H_i(WH_{64})} + P_{H_i(WH_{32})}) \\ &= k.P_{WH_{64}} + k.P_{WH_{32}} \qquad (24) \\ &= P_{BF_{21-byte}} = P_{BF_{22-byte}} = ... = P_{BF_{24-byte}} \end{aligned}$$

25-byte long input requires two WH_{64}'s. 200 bits exceed the capacity given by one WH_{64} and one WH_{32}, which adds up to 24-byte input processing. Two WH_{64}s are capable of processing 32-byte input strings, so all the Bloom filter engines operating on the range of input 25 bytes to 32 bytes long consumes the same amount of power. Again multi-hashing is used to compute hash values over such long input streams.

$$\begin{aligned} P_{BF_{25-byte}} &= \sum_{i=1}^{k} (P_{H_i(WH_{64})} + P_{H_i(WH_{64})}) = 2.k.P_{WH_{64}} \\ P_{BF_{25-byte}} &= P_{BF_{26-byte}} = ... = P_{BF_{32-byte}} \qquad (25) \end{aligned}$$

By substituting the above values into the total power consumption Equ.16, power consumption of the Bloom filter engine shown in Fig. 4 is given as

$$\begin{aligned} P_{TOTAL} &= P_{BF_{3-byte}} + P_{BF_{4-byte}} + \\ &\quad P_{BF_{5-byte}} + ... + P_{BF_{(32-byte)}} \\ P_{TOTAL} &= \underset{[3-4]}{k.P_{WH_{16}} + k.P_{WH_{16}}} + \\ &\quad \underset{[5-8]}{k.P_{WH_{32}} + ... + k.P_{WH_{32}}} + \\ &\quad \underset{[9-16]}{k.P_{WH_{64}} + ... + k.P_{WH_{64}}} + \end{aligned}$$

$$\begin{aligned} &\underset{[17-20]}{k.(P_{WH_{64}} + P_{WH_{16}}) + ... + k.(P_{WH_{64}} + P_{WH_{16}})} + \\ &\underset{[21-24]}{k.(P_{WH_{64}} + P_{WH_{32}}) + ... + k.(P_{WH_{64}} + P_{WH_{32}})} + \\ &\underset{[25-32]}{2.k.P_{WH_{64}} + ... + 2.k.(P_{WH_{64}})} \qquad (26) \end{aligned}$$

which simplifies to Equ. 27

$$P_{TOTAL} = k.(6.P_{WH_{16}} + 8.P_{WH_{32}} + 32.P_{WH_{64}}) \qquad (27)$$

B. System relying on pipelined Bloom Filters

By following a similar approach used in the regular Bloom filter analysis, we will calculate the total power consumption of the pipelined Bloom filter architecture shown in Fig. 4. The total power consumption in the system is given as in Equ. 16. Each individual power consumption of pipelined Bloom filters is calculated based on the length of the input that enters into each pipelined Bloom filter.

3-byte long input is processed by a WH_{16} owing to multi-hashing. The power consumption of a pipelined 3-byte input processing Bloom filter, $P_{PBF_{3-byte}}$, can be calculated by using Equ. 8

$$\begin{aligned} P_{PBF_{3-byte}} &= \sum_{i=1}^{r} (P_{H_i(WH_{16})}) + p \times \sum_{j=r+1}^{k} P_{H_j(WH_{16})} \\ &= r.P_{WH_{16}} + p.(k-r)P_{WH_{16}} \qquad (28) \end{aligned}$$

In order to process 4-byte input, WH_{16} is used as hash function implementation. Hence, they consume the same power as 3-byte input case.

$$P_{PBF_{3-byte}} = P_{PBF_{4-byte}} \qquad (29)$$

As for the 40 bits input, a WH_{16} can not process the input, thus a WH_{32} can handle that long input as much as 64 bits long. As a result, 5-byte long input processing Bloom filter consumes the same power as 6, 7,and 8-byte long input processing ones.

$$\begin{aligned} P_{PBF_{5-byte}} &= \sum_{i=1}^{r} (P_{H_i(WH_{32})}) + p \times \sum_{j=r+1}^{k} P_{H_j(WH_{32})} \\ &= r.P_{WH_{32}} + p \times (k-r)P_{WH_{32}} \\ P_{PBF_{5-byte}} &= P_{PBF_{6-byte}} = ... = P_{PBF_{8-byte}} \qquad (30) \end{aligned}$$

9-byte input processing can not be handled with a WH_{32}, but a WH_{64} type of hash function can handle as much as 16-byte inputs, so all the Bloom filters processing input length of 9 to 16 bytes, are using a single WH_{64} as hash function implementation.

$$\begin{aligned} P_{PBF_{9-byte}} &= \sum_{i=1}^{r} (P_{H_i(WH_{64})}) + p \times \sum_{j=r+1}^{k} P_{H_j(WH_{64})} \\ &= r.P_{WH_{64}} + p \times (k-r)P_{WH_{64}} \\ P_{PBF_{9-byte}} &= P_{PBF_{10-byte}} = ... = P_{PBF_{16-byte}} \qquad (31) \end{aligned}$$

17-byte input processing will require multi-hashing, by using WH_{64} and WH_{16}, in parallel, computes hash values on input strings of length varying from 17 to 20 bytes.

$$\begin{aligned} P_{PBF_{17-byte}} &= \sum_{i=1}^{r} (P_{H_i(WH_{64})} + P_{H_i(WH_{16})}) + \\ &\quad p \times \sum_{j=r+1}^{k} (P_{H_j(WH_{64})} + P_{H_j(WH_{16})}) \\ &= r.(P_{WH_{16}} + P_{WH_{64}}) + \\ &\quad p \times (k-r)(P_{WH_{16}} + P_{WH_{64}}) \\ P_{PBF_{17-byte}} &= P_{PBF_{18-byte}} = ... = P_{PBF_{20-byte}} \qquad (32) \end{aligned}$$

In a similar manner, 21-byte to 24-byte input processing can be done by using a WH_{64} and WH_{32} in parallel due to the principle of multi-hashing. Power consumption of 21 to 24 bytes long input processing Bloom filter is equal to each other and given by

$$\begin{aligned} P_{PBF_{21-byte}} &= \sum_{i=1}^{r} (P_{H_i(WH_{64})} + P_{H_i(WH_{32})}) + \\ &\quad p \times \sum_{j=r+1}^{k} (P_{H_j(WH_{64})} + P_{H_j(WH_{32})}) \\ &= r.(P_{WH_{32}} + P_{WH_{64}}) + \\ &\quad p \times (k-r)(P_{WH_{32}} + P_{WH_{64}}) \\ P_{PBF_{21-byte}} &= P_{PBF_{22-byte}} = ... = P_{PBF_{24-byte}} \qquad (33) \end{aligned}$$

25-byte long input requires two WH_{64} in parallel. Pipelined Bloom filters operating in range 25 bytes to 32 bytes have the same power consumption, and the power consumption is given by

$$\begin{aligned} P_{PBF_{25-byte}} &= \sum_{i=1}^{r}(2.P_{H_i(WH64)}) + \\ &\quad p \times \sum_{j=r+1}^{k}(2.P_{H_j(WH64)}) \\ &= r.2.P_{WH_{64}} + p \times (k-r)2.P_{WH_{64}} \\ P_{PBF_{25-byte}} &= P_{PBF_{26-byte}} = ... = P_{PBF_{32-byte}} \quad (34) \end{aligned}$$

Total power consumption of the system consisting of pipelined Bloom filters is given by,

$$\begin{aligned} P_{TOTAL} &= 6.[r + p(k-r)].P_{WH_{16}} + \\ &\quad 8.[r + p(k-r)].P_{WH_{32}} + \\ &\quad 32.[r + p.(k-r)].P_{WH_{64}} \\ P_{TOTAL} &= [r + p.(k-r)] \\ &\quad \times (6P_{WH_{16}} + 8P_{WH_{32}} + 32P_{WH_{64}}) \quad (35) \end{aligned}$$

By substituting *p* from Equ. 7 into the total power consumption in Equ. 35,

$$\begin{aligned} P_{TOTAL} &= \left[r + (1 - e^{\frac{-kn}{m}})^r \times (k-r)\right] \times \\ &\quad (6P_{WH_{16}} + 8P_{WH_{32}} + 32P_{WH_{64}}) \quad (36) \end{aligned}$$

After substituting the power consumption values of 16 bits, 32 bits, and 64 bits WH type of hash functions from [12], the total power consumption of a pipelined Bloom filter based *NIDS* is plotted in Fig. 5. For different values of bits per signature, $\frac{m}{n}$, total power consumption vs. number of hash functions in the first stage is illustrated. The total power consumption minimized for all of the $\frac{m}{n}$ values when the number of hash functions in the first stage is approximately 5. Before this threshold value, the match probability is such large that most of the time second stage of hash functions are used to determine the outcome of the filter, so the total power consumption increases. After the threshold value, the number of hash functions utilized in the first stage increases, which results in the increase of the overall the power consumption of the system.

IV. Conclusion

We propose to pipeline the hash functions in the Bloom filters that are used in network intrusion detection systems. Analytical results show that pipelining technique significantly decreases the total power consumption of a Bloom filter up to 90%. These type of Bloom filters are especially power efficient when the network is not malignantly congested. It is shown that the less the number of hash functions is implemented in the fist stage of a pipelined Bloom filter, the more the power consumption is. It is also true that the number of bits allocated to per signature, $\frac{m}{n}$, affects the power saving ratio in a pipelined Bloom filter. The pipelined Bloom filters are more

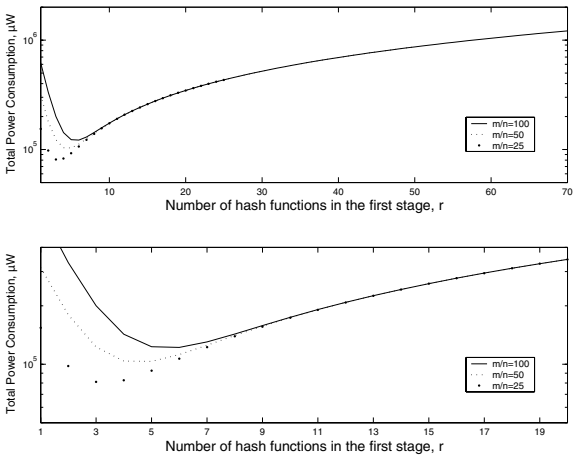

Fig. 5. Total power consumption w.r.t. the number of hash functions utilized in the first stage

appropriate when implementing network intrusion detection systems, since they are consuming less power compared to the regular Bloom filters. The selection of the hash functions to be deployed in the first stage of a pipelined Bloom filter is another crucial aspect, and it is left as a future work.

References

[1] M. Attig, S. Dharmapurikar, and J.L. Lockwood. "Implementation Results of Bloom Filters for String Matching", *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 322-323. Washington, DC., 2004.

[2] B. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors", *Commun. ACM*, vol. 13, no. 7, pp. 422-426, July 1970.

[3] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey", *Internet Mathematics*, vol. 1, no. 4, pp. 485-509, July 2003.

[4] J. L. Carter and M. Wegman, "Universal classes of hash functions", *Journal of Computer and System Sciences*, vol. 18, pp. 143-154, 1978.

[5] S. Czerwinski, B. Y. Zhao, T. Hodes, A. D. Joseph, and R. Katz. "An Architecture for a Secure Service Discovery Service", *Proc. ACM/IEEE International Conference on Mobile Computing and Networking Transactions on Networking* , pp. 24-35. New York, 1999.

[6] L. Fan, P. Cao, J. Almeida, A. Z. Broder. "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281-293, 2000.

[7] S. Dharmapurikar, P. Krishnamurthy, T.S. Sproull, and J. W. Lockwood, "Deep Packet Inspection Using Parallel Bloom Filters", *IEEE Micro*, vol. 24, no. 1, pp. 52-61, 2004.

[8] M. Mitzenmacher, "Compressed Bloom filters", *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604-612, October, 2002.

[9] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient Hardware Hashing Functions for High Performance Computers", *IEEE Transactions on Computers*, vol. 48, no. 12, pp. 1378-1381, 1997.

[10] P. Rogaway, "Bucket Hashing and Its Application to Fast Message Authentication", *Journal of Cryptology*, vol. 12, no. 2, pp. 91-116, 1999.

[11] The Sourcefire Vulnerability Research Team, "Official Snort Ruleset", Sourcefire, Inc., Columbia, MD, August 2005. (web: http://www.snort.org/pub-bin/downloads.cgi)

[12] K. Yuksel, "Universal Hashing for Ultra-Low-Power Cryptographic Hardware Applications", *MS Thesis*,Worcester Polytechnic Institute, 2004.